MyBatis

一、概述

> 框架：封装通用功能，软件开发中的半成品，简化开发过程。
>
> 轻量级的，持久层框架，负责完成java和数据库的通信。
>
> 代码分布：DAO+Service

> MyBatis 本是**apache**的一个开源项目**iBatis**, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。
>
> iBATIS一词来源于“internet”和“abatis”的组合，是一个基于Java的**持久层**框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAO）

1. 诞生背景

> Java的原生数据库通信API(jdbc),使用过于繁琐，而且随着数据变得复杂越发变得繁冗。
>
> 如下一个极其简单的查询动作，足以说明问题：

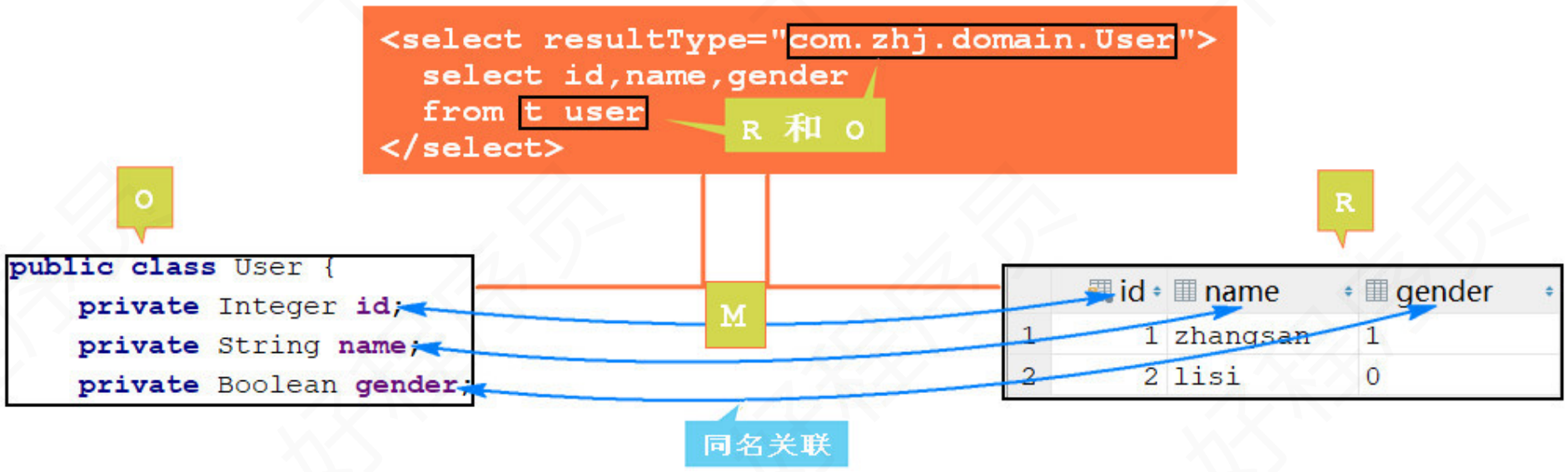
```
// 1> 自己管理 所有底层对象：驱动类，Connection，Statement,ResultSet
Class.forName("com.mysql.jdbc.Driver");
Connection connection = DriverManager.getConnection("mysql:jdbc://localhost:3306/db9?\
                                                    useUnicode=true&characterEncoding=utf8");
PreparedStatement preparedStatement = connection.prepareStatement("select id,name,age from t_user");
ResultSet resultSet = preparedStatement.executeQuery();
// 2> 自己处理数据转换过程：【 数据表格 ==> java对象 】
ArrayList<User> users = new ArrayList<User>();
while(resultSet.next()){
  Integer id = resultSet.getInt("id");
    String name = resultSet.getString("name");
    Integer age = resultSet.getInt("age");
    User user = new User(id,name,age);
    users.add(user);
}
```

2. ORM

> 概念：Object Relational Mapping，对象关系映射。
>
> 目标：在【java对象】 和 【关系表】 建立映射
>
> ：简化两者之间的通信过程。可以直接通信。(ops: 如上的过程不再用自己处理)

> 细节：ORM是持久层框架(MyBatis，Hibernate)，的重要底层设计思路。为简化持久层开发提供驱动
>
> ：java持久层框架将jdbc纳入底层，然后上层架设orm，使开发者脱离jdbc的繁琐

MyBatis对ORM的践行方式：



二、编码过程

> - Jdk环境：jdk1.8
> - 数据库环境：MySQL 5.1
> - Mybatis：3.4.5

##1. 搭建流程

1.1 导入依赖

```
// pom.xml
<!-- mybatis 依赖 -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.5</version>
</dependency>
<!-- mysql驱动 依赖 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.25</version>
</dependency>
```

```
//pom.xml,使得src/main/java下的xml文件可以进入编译范围
    <build>
        <resources>
            <resource>
                <directory>src/main/java</directory>
                <includes>
                    <include>**/*.xml</include>
                </includes>
            </resource>
            <resource>
                <directory>src/main/resources</directory>
                <includes>
                    <include>**/*.xml</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

1.2 定义O和R

```
create table t_user(
    id int primary key auto_increment,
    name varchar(20),
    gender char(1),-- tinyint(1)
    create_time datetime
)default charset=utf8 engine=innodb;
```

```
class User{
    private Integer id;
    private String name;
    private Boolean gender;
    private Date createTime;
    //get/set....
}
```

1.3 M-定义映射文件

> DAO接口定义不变，映射文件即Mapper文件替换之前的DAO实现

```
public interface UserDAO {
    public List<User> queryAll();
    public User queryOne(Integer id);
    public List<User> queryManyByName(String name);
    public List<User> queryManyByDate(Date craeteTime);
     // 明确为参数定义别名，用于mapper文件中获取
    public List<User> queryUserByIdAndName(@Param("id") Integer id, @Param("name") String name);
}
```

```
// UserDAO.xml 重点搞清楚 何为映射
<!-- 不用定义dao的实现类，此映射文件等价于dao的实现 -->
<!-- namespace="dao接口路径"-->
<?xml version="1.0" encoding="UTF-8"?>
<!-- dtd: document type definition 配置文件规范 -->
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.zhj.dao.UserDAO">
    <!--
    根据id查询用户，User queryOne(Integer id)
        select: 配置查询语句
            id: 可以通过id找到执行的statement,对应dao中的一个方法名
 parameterType: 参数类型 【int long float double string boolean date 自建类型(com.zhj.domain.User)】
    resultType:结果类型，查询的结果会被封装到对应类型的对象中
           #{}:相当于占位符
          #{id}: 就是把 “queryOne” 方法的名为id的参数填充到占位上

     -->
    <select id="queryOne" parameterType="int" resultType="com.zhj.domain.User">
        select id as id,name as name,gender as gender,create_time as createTime
        from t_user
        where id=#{id}
    </select>

    <!-- 注意: 返回类型实际是List<User> , 但此处只需定义User即可 -->
    <select id="queryAll" resultType="com.zhj.domain.User">
        select id,name,gender,create_time as createTime
        from t_user
    </select>

    <select id="queryManyByName" parameterType="string" resultType="com.zhj.domain.User">
        select id,name,gender,create_time as createTime
        from t_user
        where name like #{name}
    </select>
    <select id="queryManyByDate" parameterType="date" resultType="com.zhj.domain.User">
        select id,name,gender,create_time as createTime
        from t_user
        where create_time=#{createTime}
    </select>

    <!-- 注意: 此时方法有多个参数, 【#{xx}】取值时，需要@param支持 -->
```

```
    <select id="queryUsers" resultType="com.qianfeng.pojo.User"
            parameterType="com.qianfeng.pojo.User">
        select id,name,gender,birth from t_user where id=#{id} or name like #{name}
    </select>
</mapper>
```

1.4 搭建配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- 使用 id为“development”的环境信息 -->
    <environments default="development">
        <environment id="development">
            <!-- 配置JDBC事务控制，由mybatis进行管理 -->
            <transactionManager type="JDBC"></transactionManager>
            <!-- 配置数据源，采用mybatis连接池 -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver" />
                <!-- 【&】是特殊字符，【&amp;】是【&】的转义 -->
                <property name="url" value="jdbc:mysql://localhost:3306/db9?
useUnicode=true&amp;characterEncoding=utf8" />
                <property name="username" value="root" />
                <property name="password" value="111111" />
            </dataSource>
        </environment>
    </environments>
    <!-- 加载映射文件 -->
    <mappers>
        <!-- 使用资源的路径
        <mapper resource="com/zhj/dao/UserDAO.xml"/> -->
        <!-- 加载某个包下的映射文件 （推荐）
            要求：Mapper接口的名称与映射文件名称一致 且 必须在同一个目录下 -->
        <package name="com.zhj.dao"/>
    </mappers>
</configuration>
```

2. 测试使用

2.1 核心API

> - **`SqlSessionFactoryBuilder`**： 该对象负责加载MyBatis配置文件 并 构建SqlSessionFactory实例
> - **`SqlSessionFactory`**： 每一个MyBatis的应用程序都以一个SqlSessionFactory对象为核心。负责创建SqlSession对象实例。
> - **`SqlSession`**： 等价于jdbc中的Connection，用于完成数据操作。

```
//1、读取配置文件
String resource = "configuration.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
//2、根据配置文件创建SqlSessionFactory
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
//3、SqlSessionFactory创建SqlSession
SqlSession sqlSession = sqlSessionFactory.openSession();
```

2.2 数据操作

> - 通过SqlSession获得DAO实现（Mapper）
> - 通过DAO实现完成具体的数据操作

```
// 获得UserDAO的实现
// UserDAO userMapper  = sqlSession.getMapper(UserDAO.class);
UserDAO userDAO  = sqlSession.getMapper(UserDAO.class);
//根据id查询
userDAO.queryOne(1);
//查询所有
userDAO.queryAll();
//根据姓名模糊查询
userDAO.queryManyByName("%zhj%");
//根据日期查询
Date date = new GregorianCalendar(2019, 11, 12).getTime();
userDAO.queryManyByDate(date);
```

> 回收资源：操作最后，关闭sqlSession

```
sqlSession.close();//关闭sqlSession
```

2.3 分页操作

> 定义分页信息类：

```
class Page{
    private Integer pageNum; //页号
    private Integer pageSize; //每页几条数据
    private Integer offset; //偏移量，即从哪条查起
    //offset的get方法中动态计算偏移量
    public Integer getOffset() {
        return (pageNum-1)*pageSize;
    }
    //其他set/get
}
```

> 定义DAO接口：

```
interface User{
    ...
  public List<User> queryUserByPage(Page page);
}
```

> 定义Mapper：

```
<!-- #{offset} 取值时实际是会调用Page类中的getOffset()方法。 -->
<select id="queryUserByPage" parameterType="com.zhj.domain.Page" resultType="User">
  select id,name,gender,create_time from t_user
    limit #{offset},#{pageSize}
</select>
```

> 测试：

```
UserDAO userDAO = sqlSession.getMapper(UserDAO.class);
Page page = new Page(3,3);//第3页，每页3条
List<User> users = userDAO.queryUserByPage(page);
```

3. 增删改操作

3.1 DAO接口

```
public interface UserDAO {
    ...
    public Integer insertUser(User user);
    public Integer deleteUser(Integer id);
    public Integer updateUser(User user);
}
```

3.2 映射文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace="dao接口路径"-->
<mapper namespace="com.zhj.dao.UserDAO">
    ....
    ....
    <!-- 添加用户
        1>注意: #{方法参数对象的属性名}
        2>parameterType 和 resultType中都可以使用 缺省包
        3>增加时id补全两种方式
     -->
    <insert id="insertUser" parameterType="com.zhj.domain.User">
        <selectKey resultType="int" keyProperty="id" order="AFTER">
            SELECT last_insert_id()
        </selectKey>
        insert into tt2 (name,gender,create_time)
        values(#{name},#{gender},#{createTime})
    </insert>
    <insert id="insertUser2" parameterType="com.zhj.domain.User">
        insert into tt2 (name,gender,create_time)
        values(#{name},#{gender},#{createTime})
    </insert>

    <!--
        删除用户
     -->
    <delete id="deleteUser" parameterType="int">
        delete from tt2
        where id=#{id}
    </delete>

    <!--
        修改用户
     -->
    <update id="updateUser" parameterType="com.zhj.domain.User">
        update tt2 set name=#{name},gender=#{gender},create_time=#{createTime}
        where id= #{id}
    </update>
</mapper>
```

3.3 测试使用

> 注意：增删改操作需要控制事务，否则操作不会同步到数据库

```java
//事务控制
SqlSession sqlSession =  sqlSessionFactory.openSession();// 随着session获得，事务会自动开启
.... //数据操作
.... //数据操作
sqlSession.commit();//提交事务
sqlSession.rollback();//回滚事务
```

> 增删改操作

```java
try{
    ....
  userDAO.updateUser(user);//除了需要控制事务之外，和查询操作使用无异
    /**User u = new User(...);
    userDAO.insertUser(u);**/
    /**userDAO.deleteUser(1)**/
  sqlSession.commit();//提交事务
}catch(Exception e){
    e.printStackTrace();
    sqlSession.rollback();//回滚事务
}finally{
    if(sqlSession!=null){
      sqlSession.close();//回收资源
```

```
    }
}
```

3.4 增加细节

> ID缺失!

```
User user = new User("zhj",true,new Date());//此时user没有id
userDAO.insertUser(user);
sqlSession.commit();//此时数据已经插入到数据库中，数据库中有id，但user依然没有id
System.out.println(user.getId());//没有id
//则无法得知插入的数据是哪一条，如果后续程序需要此id，则出现bug!
```

> **两种解决方案：**
>
> 1> selectKey标签
>
> 2> useGenerateKeys 和 keyProperty属性

```
<insert id="insertUser" parameterType="com.zhj.domain.User">
  <!-- AFTER：此中语句在插入语句之后执行
        resultType="int"：此中语句执行后的返回类型是 int
        keyProperty="id"：此中语句返回值要 传递给当前方法参数中的id属性 (com.zhj.domain.User的id属性)
        select last_insert_id()：mysql特性语句，返回当前事务中，最近一次插入的数据的id-->
    <selectKey resultType="int" keyProperty="id" order="AFTER">
        select last_insert_id()
    </selectKey>
    insert into t_user (name,gender,create_time)
    values(#{name},#{gender},#{createTime})
</insert>
```

```
<!-- useGeneratedKeys="true" 声明此处添加中id用的是自动增长
     keyProperty="id" 将id值 传递给当前方法的参数中的id属性 (com.zhj.domain.User的id属性)-->
<insert id="insertUser" parameterType="com.zhj.domain.User" useGeneratedKeys="true"
        keyProperty="id">
    insert into t_user (name,gender,create_time)
    values(#{name},#{gender},#{createTime})
</insert>
```

三、别名

> 在mapper文件中， `parameterType` 和 `resultType` 中使用类型时：
>
> `<select  id="xxx" parameterType="com.zhj.domain.Page" resultType="com.zhj.domain.User">`
>
> 除mybatis自动映射的类型外，其他类型都要定义完整路径，相对繁琐。可用如下两种方式简化：

```
// configuration.xml
<configuration>
    ...
    <typeAliases>
        <!-- 1. 单独为每个类定义别名，则 "Page"等价于"com.zhj.domain.Page"
    <typeAlias type="com.zhj.domain.Page" alias="Page"/>
        <typeAlias type="com.zhj.domain.User" alias="User"/>-->
        <!-- 2. 定义缺省包，当mapper中的类型没有定义包时，使用此配置作为默认包；
       则 “Page” 会自动认为是 “com.zhj.domain”中的“Page”，即“com.zhj.domain.Page”
       则 “User” 会自动认为是 “com.zhj.domain”中的“User”，即“com.zhj.domain.User”-->
        <package name="com.zhj.domain"/>
    </typeAliases>
    ...
</configuration>
```

> 有如上别名配置后,mapper中：
>
> `<select  id="xxx" parameterType="Page" resultType="User">`

四、参数分离

> 在mybatis的配置文件中有一项重要且可能需要经常改动的配置，即，数据库连接参数。
>
> 可以单独进行管理，方便后续维护。

```
<!-- 如下四项参数 -->
<dataSource type="POOLED">
    <property name="driver" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/db9?
useUnicode=true&amp;characterEncoding=utf8"/>
    <property name="username" value="root" />
    <property name="password" value="111111" />
</dataSource>
```

1. 单独定义参数文件

```
# 在resources目录下，创建 jdbc.properties文件
# 参数名=参数值
jdbc.user=root
jdbc.password=111111
jdbc.url=jdbc:mysql://localhost:3306/db9?useUnicode=true&characterEncoding=utf8
jdbc.driver=com.mysql.jdbc.Driver
```

2. 加载参数文件

```
// 在 configuration.xml中
<configuration>
    <!-- 加载参数文件 -->
    <properties resource="jdbc.properties"></properties>
    ....
    ....
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <!-- 使用 ${参数名} 获取参数文件中的值。如此这四项参数需要修改时，需要改动参数文件即可 -->
                <property name="driver" value="${jdbc.driver}" />
                <property name="url" value="${jdbc.url}" />
                <property name="username" value="${jdbc.user}" />
                <property name="password" value="${jdbc.password}" />
            </dataSource>
        </environment>
    </environments>
```

五、关联关系

> 项目中的表都不是孤立的，会彼此关联。在操作数据时，需要联合多张表。
>
> 我们需要将表间的关系映射清楚，mybatis可以提供很好的 多表联合操作支持， 比如：
>
> 查询所有用户及其所有订单　查询所有学生及其所学课程 等操作，可以省去很多冗繁代码，直接拿到结果。

> **关联关系种类：　1对1　1对多　多对多**
>
> Person(公民)　Passport(护照)　1对1
>
> User(用户)　Order(订单)　1对多
>
> Student(学生)　Course(课程)　多对多

##1. 一对一

1.1 建表

```
drop table IF EXISTS t_passport;
drop table IF EXISTS t_person;
create table t_person(
```

```
  id int primary key AUTO_INCREMENT,
  name VARCHAR(50),
  age SMALLINT
)DEFAULT CHARSET = utf8 ENGINE =innodb;

create table t_passport(
  id int primary key AUTO_INCREMENT,
  note VARCHAR(50),
  create_time DATE,
  person_id int UNIQUE,
  FOREIGN KEY(person_id) REFERENCES t_person(id)
)DEFAULT CHARSET = utf8 ENGINE =innodb;

insert into t_person(name,age) values("zhangsan",18);
insert into t_person(name,age) values("lisi",19);
insert into t_passport(note,create_time,person_id) values("pass1",'2019-05-07',1);
insert into t_passport(note,create_time,person_id) values("pass2",'2019-05-07',2);
```

1.2 建类

```
public class Person {
    private Integer id;
    private String name;
    private Integer age;
    //关系属性
    private Passport passport;
    //set/get
}
public class Passport {
    private Integer id;
    private String note;
    private Date createTime;
    //关系属性
    private Person person;
    //set/get
}
```

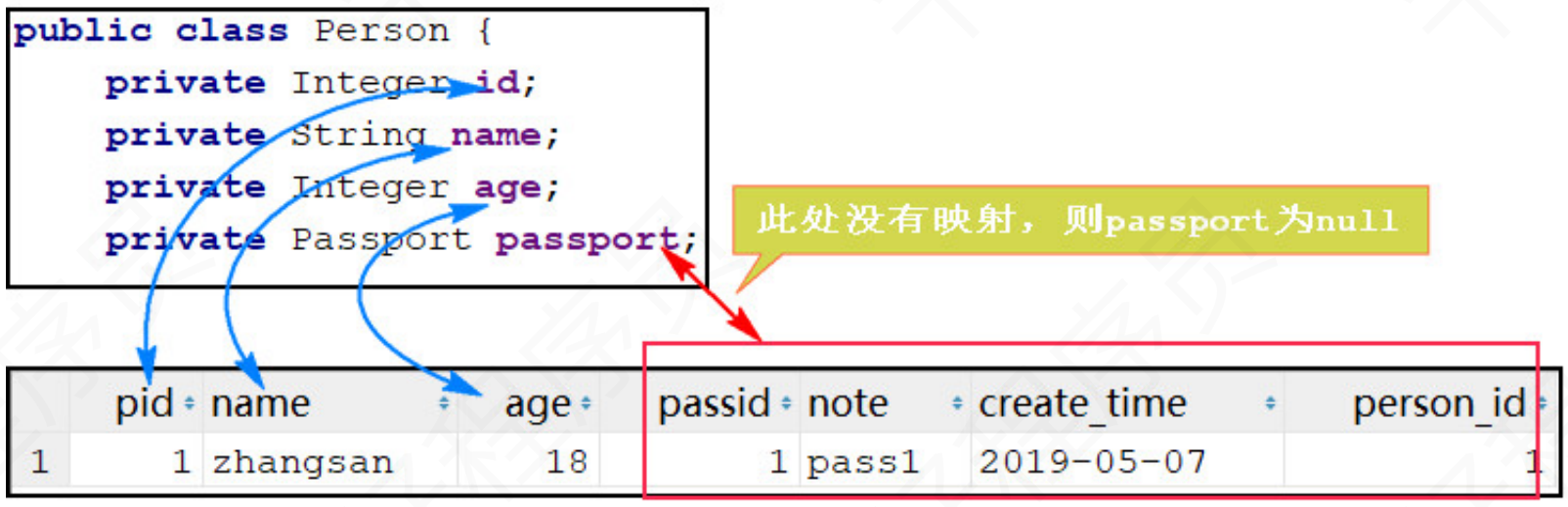
1.3 查询

> 查询某个Person，及其Passport

```
//PersonDAO ： 查询某个Person的信息及其Passport的信息
public Person queryPersonAndPassport(Integer id);
```

```
//映射文件
<select id="queryPersonAndPassport" resultType="Person" parameterType="int">
    select
        t_person.id as pid,
        name,
        age,
        t_passport.id as passid,
        note,
        create_time,
        person_id
    from t_person join t_passport
    ON t_person.id = t_passport.person_id
    where t_person.id=#{id};
</select>
```

如上映射文件，存在问题 ==> 关系属性：passport，需要多个列映射到这一个属性上，但并没有映射清楚



若要完成上图中的映射，不能再使用默认的同名映射规则，需要定制映射规则。

`resultMap` -- 映射规则定制利器

```xml
<!-- 定义查询中的各个列 与 属性的映射规则 -->
<!-- id=标识   type=返回类型(将列映射到User的属性上) -->
<resultMap id="person_passport" type="Person">
    <!-- 主键列pid，用 id标签配置； property=属性名  column=列名 -->
    <id property="id" column="pid"/>
    <!-- 常规列用 result标签配置； property=属性名  column=列名 -->
    <result property="name" column="name"/>
    <result property="age" column="age"/>
    <!-- 重点：关系属性【Passport passport】是一个对象
     用 association标签配置,如下配置就是在完成上图中 未完成之映射
         property=属性名  javaType=该属性的类型 -->
    <association property="passport" javaType="Passport">
        <!-- 将passid列 映射到 passport的id属性中 -->
        <id property="id" column="passid"/>
        <!-- 将note列 映射到 passport的note属性中 -->
        <result property="note" column="note"/>
        <!-- 将create_time列 映射到 passport属性的createTime属性中 -->
        <result property="createTime" column="create_time"/>
    </association>
</resultMap>

<!-- 删除之前的resultType， 改为使用resultMap="person_passport"
     即，采用resultMap中定义的映射规则去封装对象-->
<select id="queryPersonAndPassport" resultMap="person_passport" parameterType="int">
    select
        t_person.id as pid,
        name,
        age,
        t_passport.id as passid,
        note,
        create_time,
        person_id
    from t_person join t_passport
    ON t_person.id = t_passport.person_id
    where t_person.id=#{id};
</select>
```

1.4 测试

```java
// 查询id=1的person及其passport
Person person = personDAO.queryPersonAndPassport(1);
// 从person中获取passport
Passport passport = person.getPassport();
```

思考题 ==> 请自主设计，完成以下：

`查询所有Person及其Passport：public List<Person> queryAll();`

`查询某个Passport及其Person数据：public Passport queryPassportAndPerson(Integer id)`

细节：

> 增删改，没有任何特别需要改动的，正常定义<insert> <update> <delete> 即可

2. 一对多

2.1 建表

```
drop table IF EXISTS t_order;
drop table IF EXISTS t_user;
create table t_user( # 用户表
  id int primary key AUTO_INCREMENT,
  name VARCHAR(50),
  gender char(1),
  regist_time DATE
)DEFAULT CHARSET = utf8 ENGINE =innodb;

create table t_order( # 订单表
  id int primary key AUTO_INCREMENT,
  price VARCHAR(50),
  note VARCHAR(50),
  create_time DATE,
  user_id int,
  FOREIGN KEY(user_id) REFERENCES t_user(id)
)DEFAULT CHARSET = utf8 ENGINE =innodb;
insert into t_user(name,gender,regist_time) values('zhangsan','1','2019-12-12');
insert into t_user(name,gender,regist_time) values('lisi','0','2019-12-11');
insert into t_order(price,note,create_time,user_id) values(3000.55,'哈哈','2019-12-12',1);
insert into t_order(price,note,create_time,user_id) values(600.55,'哈哈2','2019-12-12',1);
insert into t_order(price,note,create_time,user_id) values(900.55,'呵呵2','2019-12-12',2);
insert into t_order(price,note,create_time,user_id) values(4000.55,'呵呵2','2019-12-12',2);
```

2.2 建类

```
public class User {
    private Integer id;
    private String name;
    private Boolean gender;
    private Date registTime;
    //关系属性，存储用户的多个订单
    private List<Order> orders;
    //set/get
}
public class Order {
    private Integer id;
    private BigDecimal price;
    private String note;
    private Date createTime;
    //关系属性
    private User user;
    //set/get
}
```

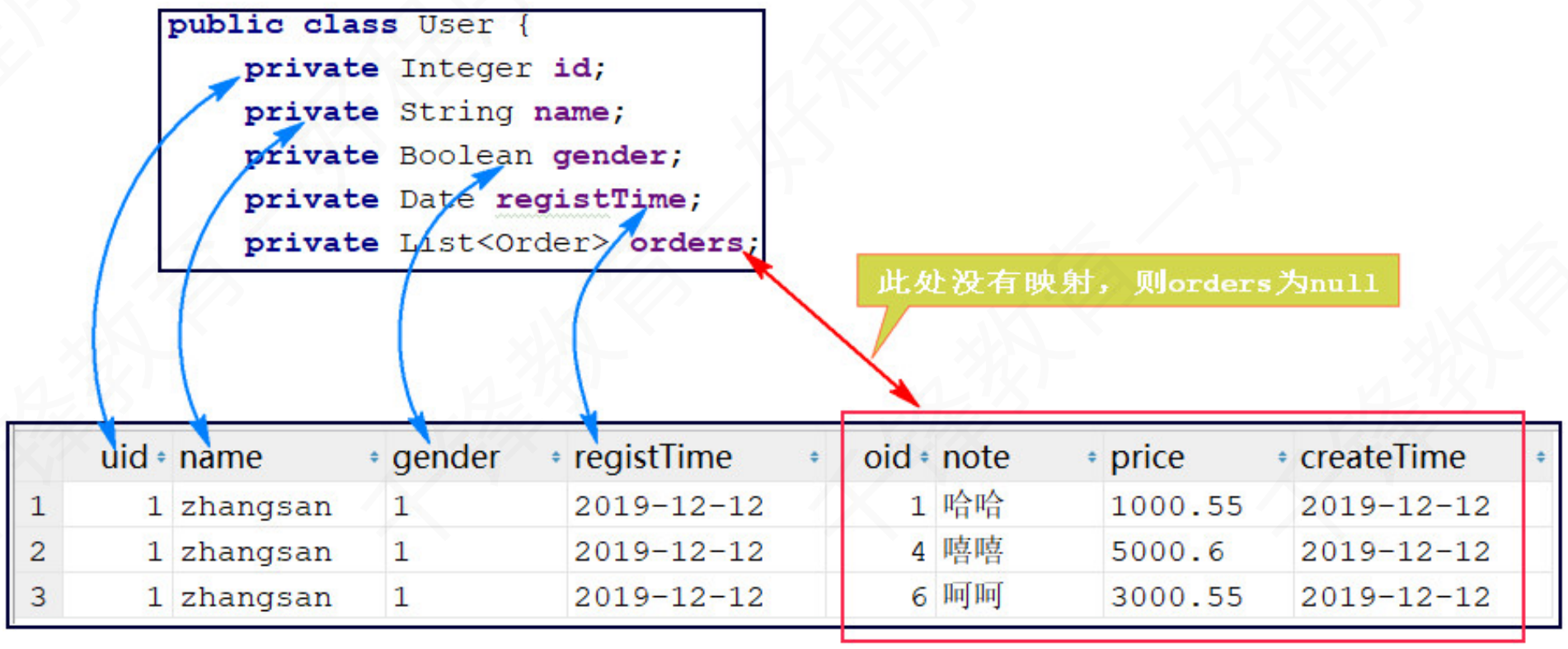
2.3 查询

```
//DAO
public queryOneUserAndOrders(Integer id);//根据用户id查询用户及其所有订单
```

```
<!-- 按照之前的思路，映射如此定义，但此时，resultType为User，所以只能接收User本身的信息！
     那如何将查询中得到的Order信息也保存到User中呢?

   User中保存Order信息的属性名为“orders”，则此属性需要有一个和它同名的列才可以接收值！
   但是该属性的值又不可能来自某一个列，而是多个列的值！

   我们该如何配置，才可以将多列的值对应到一个属性上呢?
-->
<select id="queryOneUserAndOrders" parameterType="int" resultType="User">
  select t_user.id as uid,
      name,
      gender,
        regist_time as registTime,
```

```
        t_order.id as oid,
         note,price,
         create_time as createTime
    from t_user join t_order
    ON t_user.id = t_order.user_id
    where t_user.id=#{id}
</select>
```

如上映射文件，存在问题 ==> 关系属性：orders，需要多个列映射到这一个属性上，但并没有映射清楚



	uid	name	gender	registTime	oid	note	price	createTime
1	1	zhangsan	1	2019-12-12	1	哈哈	1000.55	2019-12-12
2	1	zhangsan	1	2019-12-12	4	嘻嘻	5000.6	2019-12-12
3	1	zhangsan	1	2019-12-12	6	呵呵	3000.55	2019-12-12

若要完成上图中的映射，不能再使用默认的同名映射规则，需要定制映射规则。

`resultMap` -- 映射规则定制利器

```
<!-- 定义查询中的各个列 与 属性的映射规则 -->
<!-- id=标识    type=最终返回类型(将列映射到User的属性上) -->
<resultMap id="user_orders" type="User">
    <!-- 主键列单独用id标签标识
       property=类中属性名  column=查询中的列名-->
    <id property="id" column="uid"/>
    <!-- 常规属性都用result标签 -->
    <result property="name" column="name"/>
    <result property="gender" column="gender"/>
    <result property="registTime" column="registTime"/>
    <!-- 重点：关系属性【List<Order> orders】,非主键也非常规属性，而是一个List
         用collection标签配置，用于完成上图中未完成之映射。
         property=属性名   ofType=集合中的泛型类型-->
    <collection property="orders" ofType="Order">
        <!-- Order的主键列 -->
        <id property="id" column="oid"/>
        <!-- 常规属性用result标签 -->
        <result property="note" column="note"/>
        <result property="price" column="price"/>
        <result property="createTime" column="createTime"/>
    </collection>
</resultMap>
<!-- 此处是resultType换为resultMap,进而引用了如上定义的“user_orders” -->
<select id="queryOneUserAndOrders" parameterType="int" resultMap="user_orders">
    select t_user.id as uid,
      name,
      gender,
        regist_time as registTime,
       t_order.id as oid,
        note,price,
        create_time as createTime
    from t_user join t_order
    ON t_user.id = t_order.user_id
    where t_user.id=#{id}
</select>
```

2.4 测试

```
//查询用户1，及其订单信息
User user = userDAO.queryOneUserAndOrders(1);
//获取用户1的订单信息
List<Order> orders = user.getOrders();
```

> 思考题 ==> 请自主设计，完成以下：
>
> **查询所有User及其Order：public List<User> queryAll();**
>
> **查询某个Order及其User：public Order queryOrderAndUser(Integer orderid)**

> 细节：
>
> 增删改，没有任何特别需要改动的，正常定义<insert> <update> <delete> 即可

3. 多对多

3.1 建表

```
drop table IF EXISTS t_student;
drop table IF EXISTS t_course;
drop table IF EXISTS t_student_course;

create table t_student(
  id int primary key AUTO_INCREMENT,
  name VARCHAR(50),
  age SMALLINT
)DEFAULT CHARSET = utf8 ENGINE =innodb;

create table t_course(
  id int primary key AUTO_INCREMENT,
  title VARCHAR(50)
)DEFAULT CHARSET = utf8 ENGINE =innodb;

create table t_student_course(
  student_id int,
  course_id int,
  FOREIGN KEY (student_id) REFERENCES t_student(id),
  FOREIGN KEY (course_id) REFERENCES t_course(id),
  PRIMARY KEY (student_id,course_id)
)DEFAULT CHARSET = utf8 ENGINE =innodb;

insert into t_student (name,age) values("zhangsan",18);
insert into t_student (name,age) values("lisi",19);
insert into t_course (title) values("java基础");
insert into t_course (title) values("mybatis");
insert into t_student_course values(1,1);
insert into t_student_course values(1,2);
insert into t_student_course values(2,1);
insert into t_student_course values(2,2);
```

3.2 建类

```
public class Student {
    private Integer id;
    private String name;
    private Integer age;
    //关系属性
    private List<Course> courses;
    //set/get
}
public class Course {
    private Integer id;
    private String title;
    //关系属性
    private List<Student> students;
    //set/get
}
public class StudentCourse {
```

```
    private Integer studentId;
    private Integer courseId;
    //set/get
}
```

3.3 查询

```
//StudentDAO
public List<Student> queryStudents();
```

```
//StudentDAO.xml 映射文件
<!-- 定义映射规则 和一对多中使用一样-->
<resultMap id="student_course" type="Student">
    <id column="sid" property="id"/>
    <result column="name" property="name"/>
    <result column="age" property="age"/>
    <!-- 此处使用collection和一对多中一样 -->
    <collection property="courses" ofType="Course">
        <id column="cid" property="id"/>
        <result column="title" property="title"/>
    </collection>
</resultMap>
<!-- 查询，并根据映射规则完成数据封装 -->
<select id="queryStudents" resultMap="student_course">
    select
        t_student.id as sid,
        name,
        age,
        t_course.id as cid,
        title
    from t_student join t_student_course
    ON t_student.id = t_student_course.student_id
    JOIN t_course
    ON t_student_course.course_id = t_course.id;
</select>
```

3.4 测试

```
StudentDAO studentDAO = sqlSession.getMapper(StudentDAO.class);
//查询所有学生及其所有课程
List<Student> students = studentDAO.queryStudents();
//获取每个学生的所有课程
for(Student stu:students){
    System.out.println(stu);
    for(Course cour:stu.getCourses()){
        System.out.println(cour);
    }
}
```

3.5 增加测试

```
//StudentDAO.xml
<mapper namespace="com.zhj.dao.StudentDAO">
    <!-- 增加Student到 t_student表                    ( ops:后续操作需要新增学生的id ) -->
    <insert id="insertStudent" parameterType="Student" useGeneratedKeys="true" keyProperty="id">
        insert into t_student(name,age) values(#{name},#{age})
  </insert>
</mapper>
```

```
//StudentCourseDAO.xml
<mapper namespace="com.zhj.dao.StudentCourseDAO">
  <!-- 为学生添加课程 或 为课程添加学生 -->
    <insert id="insertStudentCourse" parameterType="StudentCourse">
        insert into t_student_course(student_id,course_id) values(#{studentId},#{courseId})
    </insert>
</mapper>
```

```
//添加学生 赵六，并为其关联课程 1
// 获得 学生DAO 和 学生课程DAO
StudentDAO studentDAO = sqlSession.getMapper(StudentDAO.class);
StudentCourseDAO scDAO = sqlSession.getMapper(StudentCourseDAO.class);
// 新建学生，并添加
Student student = new Student(null, "赵六", 22);
studentDAO.insertStudent(student);
// 新建学生赵六和课程1的关系，并添加 (为赵六添加课程 1)
StudentCourse sc = new StudentCourse(student.getId(),1);
scDAO.insertStudentCourse(sc);
// 提交
sqlSession.commit();
```

4. 关系的方向（了解）

> 单向关系：A B 双方，A中有B，或B中有A
>
> 双向关系：A B双方，A中有B，且B中有A

```
//互相持有彼此，即为双向关系
class User{
    ...
    private List<Order> orders;
}
class Order{
    ...
  private User user;
}
```

```
//Order不持有User，或User不持有Order，即为单向关系
class User{
    ...
    private List<Order> orders;
}
class Order{
    ...
}
```

六、延迟加载

> 延迟加载：不会立即查询，而是实际使用数据时再查询
>
> `association` 和 `collection` 都是在加载关系对方的数据，但在查询一方时，不一定需要另一方数据，此时可以利用延迟加载特性 做出适应。

```
// 找到mysql的 my.ini文件，修改配置，将mysql日志输出到本地磁盘，用于观测mysql实际执行了哪些查询动作
general-log=1
general_log_file="d:/LAPTOP-BT26GJ86.log"
```

1. association

> 重点：`<association  selecct=xxx   column=xxx   fetchType=xx >`

```
<mapper namespace="com.zhj.dao.PersonDAO">
    <resultMap id="person_passport" type="Person">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <result property="age" column="age"/>
        <!--<association property="passport" javaType="Passport">-->
        <!--<id property="id" column="passid"/>-->
        <!--<result property="note" column="note"/>-->
        <!--<result property="createTime" column="create_time"/>-->
        <!--</association>-->
```

```
    <!-- 重点：因为查询中并没有查询passport，所以此处没有映射 id、result等。而是用了select
         select="一个可以查询passoport的方法:queryOne"
         column="pid"是将本次查询的pid列的值传递给方法:queryOne
   fetchType="lazy" 延迟加载-->
    <association property="passport" column="id" select="com.zhj.dao.PassportDAO.queryOne"
 fetchType="lazy"/>
</resultMap>
<!-- 重点：因为此处的Passport要延迟加载，所以，并没有查询passport -->
<select id="queryPersonAndPassport" parameterType="int" resultMap="person_passport">
    <!--select
          t_person.id as pid,
          name,
          age,
          t_passport.id as passid,
          note,
          create_time,
          person_id
          from t_person join t_passport
          ON t_person.id = t_passport.person_id-->
        select
            id,
            name,
            age
        from t_person
        where id = #{id}
  </select>
</mapper>
```

```
<mapper namespace="com.zhj.dao.PassportDAO">
    <select id="queryOne2" parameterType="int" resultType="Passport">
        select
            id,
            note,
            create_time,
            person_id
        from t_passport
        where person_id=#{personId}
    </select>
</mapper>
```

```
//测试
Person person = personDAO.queryPersonAndPassport(1);//此时只执行了对person表的查询
Passport passport = person.getPassport();//此时，真正使用了Passport，才会查询passport(passport是延迟加载的)
```

2. collection

> 重点，同上

```
<mapper namespace="com.zhj.dao.UserDAO">
    <resultMap id="user_orders" type="User">
        <id property="id" column="id"/>
        <result property="name" column="name"/>
        <result property="gender" column="gender"/>
        <result property="registTime" column="registTime"/>
        <!--<collection property="orders" ofType="Order">-->
        <!--<id property="id" column="oid"/>-->
        <!--<result property="note" column="note"/>-->
        <!--<result property="price" column="price"/>-->
        <!--<result property="createTime" column="createTime"/>-->
        <!--</collection>-->
        <collection property="orders" select="com.zhj.dao.OrderDAO.queryOrderOfUser"
                    column="id" fetchType="lazy"></collection>
    </resultMap>

    <select id="queryOne" parameterType="int" resultMap="user_orders">
        <!--select t_user.id as uid,name,gender,t_user.regist_time as registTime,
                   t_order.id as oid,note,price,t_order.create_time as createTime
            from t_user join t_order
            ON t_user.id = t_order.user_id
```

```
            where t_user.id=#{id}-->
        select id,name,gender,regist_time as registTime
        from t_user
        where t_user.id=#{id}
    </select>
</mapper>
```

```
<mapper namespace="com.zhj.dao.orderDAO">
    <select id="queryOrderOfUser" parameterType="int" resultType="Order">
        select t_order.id as oid,note,price,t_order.create_time as createTime
        from t_order
        where user_id = #{userId}
    </select>
</mapper>
```

```
//测试
User user = userDAO.queryOne(1);//此时只查询了用户表，尚未查询order表
List<Order> orders = user.getOrders();//此时真正使用了order，才会查询order表（Order是延迟加载的）
```

> 会触发查询的方法有： `toString, hashCode, equals, clone`

七、#和$区分

```
//如果用$,则必须用 @Param注解，否则${name}会认为是要从参数中取名为name的属性
public List<User> test3(@Param("name") String a);
```

```
<!-- 注意${} 就是在做字符拼接，所以此处用了【'${name}'】而不是【${name}】
     类比jdbc的sql语句的拼接：
        String name="zhj";
        String sql = "select ... from tt2 where name='"+name+"'";//此处是要加单引号的
     注意：sql拼接时，有sql注入的风险
-->
<select id="test3" parameterType="string" resultType="com.zhj.domain.User">
    select id,name,gender,create_time as createTime
    from tt2
    where name = '${name}'
</select>
```

```
<!-- 注意#{} 就是在做占位符，所以此处用了【#{name}】而不是【'#{name}'】
     类比jdbc的sql语句的拼接：
        String name="zhj";
        String sql = "select ... from tt2 where name=?";//此处是不加单引号的
    ...
    prepareStatement.executeQuery(sql,name);//占位符赋值
-->
<select id="test3" parameterType="string" resultType="com.zhj.domain.User">
    select id,name,gender,create_time as createTime
    from tt2
    where name = #{name}
</select>
```

> **必须使用$场景：**

```
<select id="test3" parameterType="string" resultType="com.zhj.domain.User">
    select id,name,gender,create_time as createTime
    from tt2
    order by id ${name}
</select>
<select id="test4" parameterType="string" resultType="com.zhj.domain.User">
    select id,name,gender,create_time as createTime
    from ${tn}
    where name = #{name}
</select>
<!-- 用在列名上亦可 -->
```

```
userDAO.test3("desc");
userDAO.test3("asc");
userDAO.test4("t_user");
userDAO.test4("t_admin");
```

八、动态sql

> 在映射文件中，定义了要执行的sql语句，mybatis支持在sql语句中填充一些逻辑，是的sql语句可以呈现不同的语义，
>
> 即动态sql

1. IF

> 在sql中 注入 `if` ,可以让sql更加灵活，让一个查询语句，可以应对更多查询情景。
>
> 重点：== != > < >= <= and or
>
> :比较字符串需要加引号，比较数字、布尔、null不用加引号
>
> 常用：对参数是否为空的判断，动态决定sql语句的组成
>
> 情景：对用户可以有通过name 或 gender的搜索，如果没有 `if` 动态逻辑，则是要定义多个 `<select>`

```
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user
    WHERE
    <if test="name != null and name!=''">
        name=#{name}
    </if>
    <if test="gender != null">
        AND gender=#{gender}
    </if>
</select>
<!-- 如上如果gender为null, name不为null, 则sql为:
     SELECT id,name,gender,regist_time
    FROM t_user
    WHERE name=#{name}
-->
<!-- 如上如果gender不为null, name不为null, 则sql为:
     SELECT id,name,gender,regist_time
    FROM t_user
    WHERE name=#{name} AND gender=#{gender}
-->
```

```
<!-- 补充: 比较的其他用法 -->
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user
    WHERE
    <if test="name == null or name=='zhj'">
        name=#{name}
    </if>
    <if test="id>=1">
        AND id>#{id}
    </if>
    <if test="gender == false">
        AND gender=#{gender}
    </if>
</select>
```

2. Choose

> 如果在多个判断中，只会有一个是可以成立的，可以使用Choose标签

```
<!-- 如果id不为空就按id查询。如果id为空但name不为空就按name查询。如果都为空，就查询所有男性用户。 -->
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user
    WHERE
    <choose> <!-- 从上到下，有任何一个when标签，判断成功则停止判断 -->
        <when test="id != null">  <!-- 判断 -->
            id > #{id}
        </when>
        <when test="name !=null"> <!-- 判断 -->
            name = #{name}
        </when>
        <otherwise>  <!-- 以上判断都不成立时，执行 -->
            gender = '1'
        </otherwise>
    </choose>
</select>
```

> 注意，在sql中的判断，本意不在于判断，而在于将sql动态的适应不同场景，简化开发。

3. Where

> 动态sql在使用中，存在一些问题：

```
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time FROM t_user
    WHERE
    <if test="name != null">
        name=#{name}
    </if>
    <if test="id>=1">
        AND id>#{id}
    </if>
</select>
<!-- 如果 name=null,id=3,则sql变为:
     SELECT id,name,gender,regist_time FROM t_user
     WHERE AND id>#{id}
-->
<!-- 如果 name=null,id=0,则sql变为:
     SELECT id,name,gender,regist_time FROM t_user
     WHERE
-->
```

> `<where>` 标签中如果没有成立的条件，则不会拼接 `where语句` ；
>
> `<where>` 标签中如果以 and 或 or开头，会去将其去除。

```
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user
    <where>
        <if test="name == null">
            name=#{name}
        </if>
        <if test="id>=1">
            AND id>#{id}
        </if>
    </where>
</select>
<!-- 如果 name=null,id=3,则sql变为:
     SELECT id,name,gender,regist_time FROM t_user
     WHERE id>#{id}
-->
<!-- 如果 name=null,id=0,则sql变为:
     SELECT id,name,gender,regist_time FROM t_user
-->
```

4. Set

> `<set>` 标签主要用于 `update` 中

```
<!-- 只更新非空的字段 -->
<update id="updateUser" parameterType="User">
    UPDATE t_user2
    SET
    <if test="name != null">
        name = #{name},
    </if>
    <if test="gender != null">
        gender = #{gender},
    </if>
    <if test="registTime != null">
        regist_time = #{registTime}
    </if>
    WHERE id = #{id}
</update>
<!-- 如果registTime=null,name或gender!=null, 则sql为:
   UPDATE t_user2
     SET name = #{name},gender = #{gender},
     WHERE id = #{id}
   注意: 多了逗号, sql语法错误
-->
```

> 使用 `<set>`:

```
<update id="updateUser" parameterType="User">
    UPDATE t_user2
    <set>
        <if test="name != null">
            name = #{name},
        </if>
        <if test="gender != null">
            gender = #{gender},
        </if>
        <if test="registTime != null">
            regist_time = #{registTime}
        </if>
    </set>
    WHERE id = #{id}
</update>
<!-- 如果registTime=null,name或gender!=null, 则sql为:
   UPDATE t_user2
     SET name = #{name},gender = #{gender}
     WHERE id = #{id}
   注意: <set>会自动补充一个 "set语句", 并将最后一个逗号去除
-->
```

5. Trim

```
<select id="queryUsers" parameterType="User" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user2
    <trim prefix="where" prefixOverrides="and|or">
        <if test="name != null">
            name=#{name}
        </if>
        <if test="id>=10">
            OR id>#{id}
        </if>
        <if test="gender == false">
            AND gender=#{gender}
        </if>
    </trim>
</select>
<!-- <trim prefix="where" prefixOverrides="and|or">  等价于 <where>
```

```
     prefix="where": 会添加一个where关键字
     prefixOverrides="and|or": 会去除开头的and 或 or
-->
```

```
<update id="updateUser" parameterType="User">
    UPDATE t_user2
    <trim prefix="set" suffixOverrides=",">
        <if test="name != null">
            name = #{name},
        </if>
        <if test="gender != null">
            gender = #{gender},
        </if>
        <if test="registTime != null">
            regist_time = #{registTime}
        </if>
    </trim>
    WHERE id = #{id}
</update>
<!-- <trim prefix="set" suffixOverrides=","> 等价于 <set>
     prefix="set": 会添加一个set关键字
   suffixOverrides=",": 会去除最后的逗号
-->
```

6. Foreach

> 批量查询：id为 1,3,5的数据

```
//DAO声明为:  public List<User> queryUsers2(List<Integer> ids);
<select id="queryUsers2" resultType="User">
    SELECT id,name,gender,regist_time
    FROM t_user2
    WHERE id IN
    <foreach collection="list" open="(" separator="," close=")" item="id" index="ind">
        #{id}
    </foreach>
</select>
<!--
  <foreach collection="list" open="(" separator="," close=")" item="id" index="ind">
    collection="list" 代表参数是List, 如果是数组, 要用array
    open="("   以“(” 开头
  close=")"  以“)” 结尾
    separator=","   值之间用“,”分隔
    item="id"   每次遍历的值的临时变量
    #{id}   获取每次遍历的值
    如上: 标签的效果是 (值1,值2,值3)
    示例: 如果传入 List{1 3 5},则最终的sql:
     【SELECT id,name,gender,regist_time
      FROM t_user2
      WHERE id IN (1,3,5)】
-->
```

> 批量添加：
>
> insert into t_user (name,gender) values('xx',1),('xxx',0),('xxxx',1)

```
<!-- 批量添加 -->
<insert id="insertUsers" parameterType="java.util.List">
    insert into t_user2 (name,gender,regist_time) values
    <foreach collection="list" item="user" index="ind" close="" open="" separator=",">
        (#{user.name},#{user.gender},#{user.registTime})
    </foreach>
</insert>
```

```
List<User> users = ...;
System.out.println(userDAO.insertUsers(users));
```

7. Sql

> 复用sql语句

```
<!-- 定义一段sql -->
<sql id="order">
    id,note,price,create_time as createTime
</sql>
<!-- 引用sql -->
<select id="queryOrder" parameterType="int" resultType="Order">
    select
    <include refid="order"/>
    from t_order
    where id = #{id}
</select>
```

九、缓存

> 缓存：将数据库的数据临时的存储起来，以更好的支持查询。
>
> 问题：如果有数据，查询频繁且更新极少，此种数据如果依然每次到数据库查询，效率偏低。
>
> 解决：将如上数据，临时的存储到内存中，提供对外界的查询服务，进而减少和数据库的通信，提高查询效率。
>
> 原理：当查询数据时，查询结果会被缓存在某个内存区域中，核心存储结构={sql：查询结果}；
>
> 每次发起查询时，会先找到缓存，从中尝试获取数据，如果没有找到数据，再去查数据库，并将在数**
>
> 库中查到的结果存入缓存，以供后续查询使用。

> MyBatis作为持久层框架，缓存管理自然是他的本职工作。
>
> 支持了两种缓存： **一级缓存，二级缓存**

1. 一级缓存

> 存储位置：SqlSession；即一个SqlSession对象发起查询后，查询结果会缓存在自己内部
>
> 有效范围：同一个SqlSession的多次查询；即，同一个SqlSession的多次相同sql的查询可以使用一级缓存
>
> 开启：不用任何配置，默认启动。
>
> 清除： `sqlSession.clearCache();`

2. 二级缓存

2.1 概述

> 存储位置：SqlSessionFactory；同一个SqlSessionFactory创建的所有SqlSession发起的查询，查询结果都会缓存在
>
> SqlSessionFactory内部。
>
> 有效范围：同一个SqlSessionFactory
>
> 开启：默认开启，但需要制定哪个DAO的Mapper需要使用二级缓存，定义一个 `<cache>` 即可
>
> 注意：二级缓存必须在 `sqlSession.commit()` 或 `sqlSession.close()` 之后才生效
>
> 清除： `sqlSession.rollback();//则查询的结果不会进入二级缓存`

2.2 应用

> 二级缓存使用：

```xml
<mapper namespace="com.zhj.dao.UserDAO">
    <!-- 当前mapper中的所有查询，都进入二级缓存
         缓存数据中涉及的pojo一定要实现  Serialiable。
    -->
    <cache></cache>
    <select>...</select>
    .....
</mapper>
```

```
UserDAO userDAO1 = sqlSession1.getMapper(UserDAO.class);
UserDAO userDAO2 = sqlSession2.getMapper(UserDAO.class);
userDAO1.queryOne(1);
userDAO2.queryOne(1);
// 在开启了二级缓存的情况下，如上代码依然会查询两次数据库。
// userDAO1.queryOne(1);之后缓存只在sqlSession1中,并未进入二级缓存。userDAO2.queryOne(1);无法使用
```

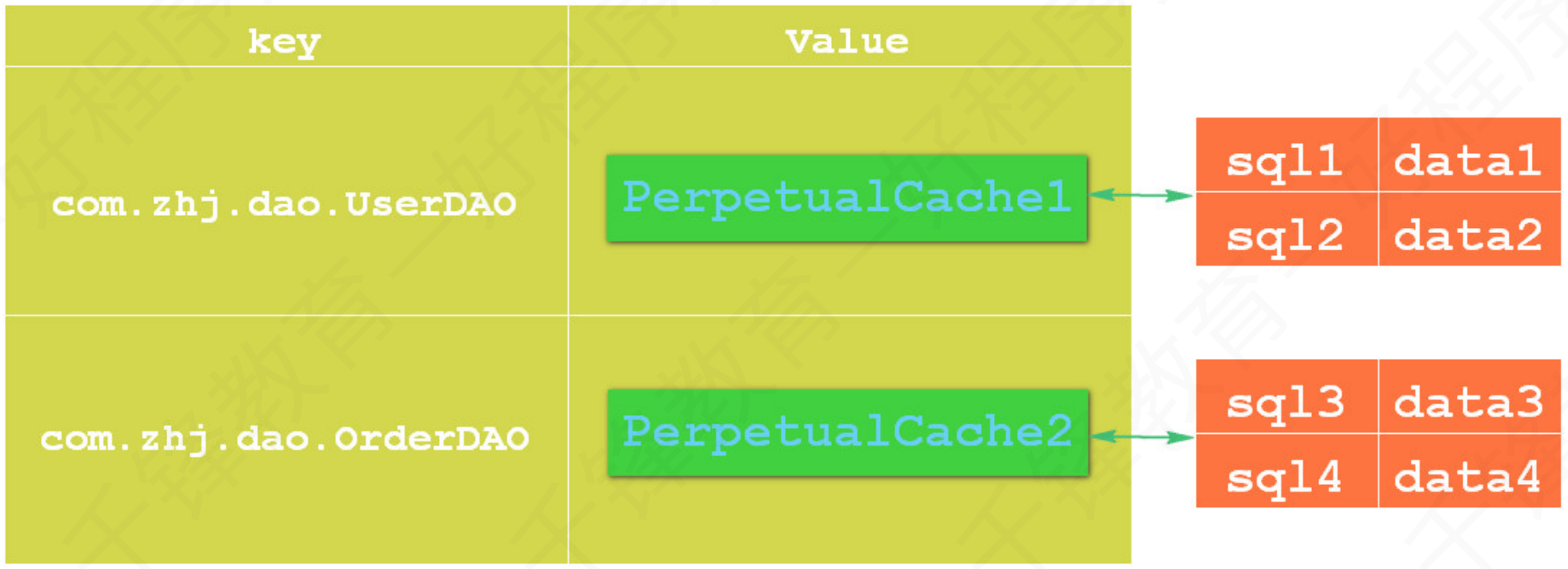
```
UserDAO userDAO1 = sqlSession1.getMapper(UserDAO.class);
UserDAO userDAO2 = sqlSession2.getMapper(UserDAO.class);
userDAO1.queryOne(1);
sqlSession1.commit();//close()也可以，因为close内部流程和commit内部流程有对缓存的相同处理
userDAO2.queryOne(1);
// 此时如上代码会只查询一次数据库。
// sqlSession1.commit();执行时，会将查到的数据序列化，存入二级缓存中。userDAO2.queryOne(1)可以使用
```

> **思考题：如下代码会如何查询数据库？**

```
SqlSession sqlSession = sqlSessionFactory.openSession();
SqlSession sqlSession2 = sqlSessionFactory.openSession();
SqlSession sqlSession3 = sqlSessionFactory.openSession();

UserDAO userDAO= sqlSession.getMapper(UserDAO.class);
UserDAO userDAO2= sqlSession2.getMapper(UserDAO.class);
UserDAO userDAO3= sqlSession3.getMapper(UserDAO.class);

userDAO.queryOne(1);
userDAO.queryOne(2);
sqlSession.close();

userDAO2.queryOne(3);
sqlSession2.close();

userDAO3.queryOne(1);
userDAO3.queryOne(2);
userDAO3.queryOne(3);
//请分析如上查询，会触发哪些数据库查询
```

2.3 结构

二级缓存存储结构



2.4 清除

> 二级缓存是以 `namespace` 为单位组织的，当某个 `namespace` 中发生数据改动，则 `namespace` 中缓存的所有数据会被mybatis清除。

```
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsReader("mybatis_config.xml"));

SqlSession sqlSession = sqlSessionFactory.openSession();
SqlSession sqlSession2 = sqlSessionFactory.openSession();
SqlSession sqlSession3 = sqlSessionFactory.openSession();
UserDAO userMapper = sqlSession.getMapper(UserDAO.class);
UserDAO userMapper2 = sqlSession2.getMapper(UserDAO.class);
UserDAO userMapper3 = sqlSession3.getMapper(UserDAO.class);

userMapper.queryUserById(1);
sqlSession.close();//二级缓存生效

// userMapper中的所有二级缓存被清除
userMapper2.updateUser(new User(1,"zs",true,new Date()));
sqlSession2.commit();

// 再次查询，二级缓存中已没有数据，会查询数据库
userMapper3.queryUserById(1);
```

2.5 cache-ref

> 和关系属性相关
>
> 注意如果 `<collection>` 中没有使用 `select` 关联查询，则不存在此问题。

```
<mapper namespace="com.zhj.dao.UserDAO">
    <cache/>
    <resultMap id="user_orders" type="User">
        <id property="id" column="uid"/>
        <result property="name" column="name"/>
        <result property="gender" column="gender"/>
        <result property="registTime" column="registTime"/>
        <collection property="orders" select="com.zhj.dao.OrderDAO.queryOrderOfUser" column="id"
            fetchType="eager/lazy"/>
    </resultMap>
    <select id="queryOne" parameterType="int" resultMap="user_orders">
        select id,name,gender,regist_time as registTime
        from t_user
        where id=#{id}
    </select>
</mapper>
```

```
<mapper namespace="com.zhj.dao.OrderDAO">
    <!-- 使用cache-ref 则OrderDAO的缓存数据，会存放于com.zhj.dao.UserDAO分支下，
       与UserDAO的缓存数据存储于同一个Perpetual对象中
    -->
    <cache-ref namespace="com.zhj.dao.UserDAO"/>
    <select id="queryOrderOfUser" parameterType="int" resultType="Order">
        select id as oid,note,price,create_time as createTime
        from t_order
        where user_id = #{userId}
    </select>
</mapper>
```

```
UserDAO userDAO = sqlSession1.getMapper(UserDao.class);
//User 和 关系属性Order 都被缓存
User user = userDAO.queryOne(1);
user.getOrders().size();
sqlSession.commit();

OrderDAO orderDAO = sqlSession2.getMapper(OrderDAO.lass);
orderDAO.queryOrderOfUser(1); //有二级缓存数据可用 (OrderDAO中必须有 <cache>或<cache-ref>)
```

```java
UserDAO userDAO = sqlSession1.getMapper(UserDao.class);
userDAO.queryOne(1);
//数据改动，此时会清空User缓存，并清空Order缓存(因为order中是 <cache-ref>,如果是<cache>则此处只会清空User缓存)
userDAO.insertUser(new User(null,"new_user",true,new Date()));
sqlSession.commit();

OrderDAO orderDAO = sqlSession2.getMapper(OrderDAO.lass);
orderDAO.queryOrderOfUser(1); //重新查询数据库
```

十、PageHelper

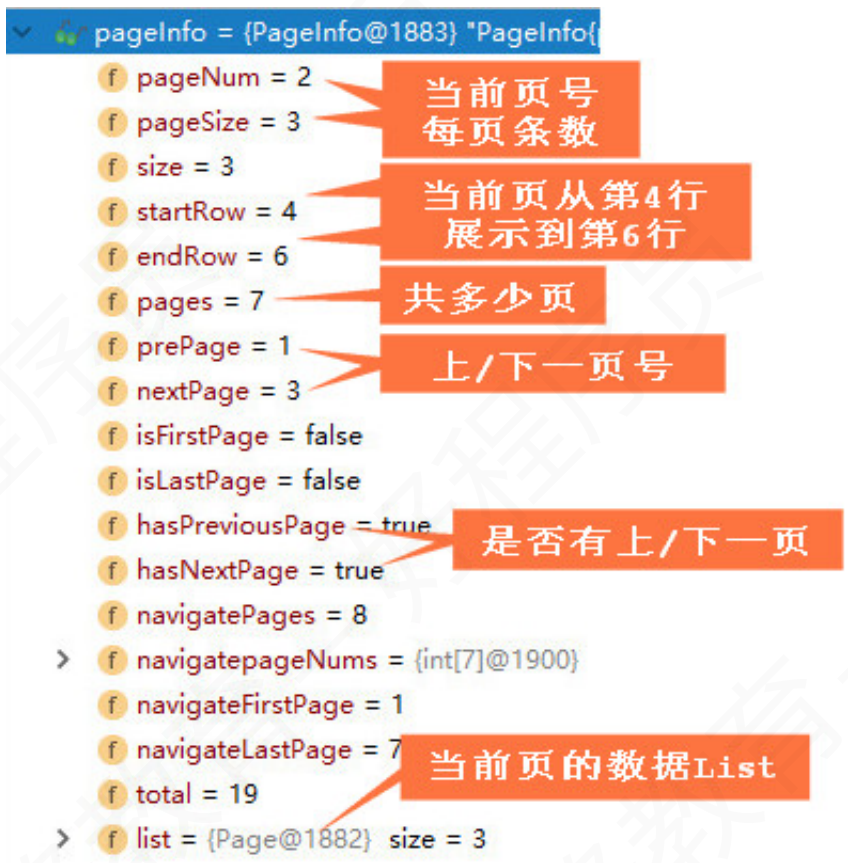
1. 使用过程

```xml
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>RELEASE</version>
</dependency>
```

```xml
<!--
    plugins在配置文件中的位置必须符合要求，否则会报错，顺序如下：
    properties?, settings?,
    typeAliases?, typeHandlers?,
    objectFactory?,objectWrapperFactory?,
    plugins?,
    environments?, databaseIdProvider?, mappers?
-->
<plugins>
    <!-- com.github.pagehelper为PageHelper类所在包名 -->
    <plugin interceptor="com.github.pagehelper.PageInterceptor">
        <!-- 页号自动回归到合理数值 -->
        <property name="reasonable" value="true"/>
    </plugin>
</plugins>



<!-- spring等价配置
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="plugins">
    <array>
      <bean class="com.github.pagehelper.PageInterceptor"></bean>
    </array>
  </property>
</bean>
-->
```

```java
//使用：
PageHelper.startPage(2,3);// 第2页，每页3条数据,pageNum,pageSize
PageHelper.orderBy("id desc");//可以选择设置排序（可选）
List<User> users = mapper.queryAllUsers();//PageHelper后的第一个查询语句，会被PageHelp增强处理(可观测mysql日志)
for (User user : users) {// users中已经是分页数据
    System.out.println(user);
}
//包装一个PageInfo，其中会持有所有分页会用到的信息：当前页号，每页多少条，共多少页，是否为第一页/最后一页，是否有下一页等。
PageInfo<User> pageInfo=new PageInfo<User>(users);
```

PageInfo对象 概览



注意：如果是多表查询，则会有如下效果：

```
select t_user.id,name,gender,regist_time,
            t_order.id orderId,price,note,create_time
        from t_user JOIN t_order
                ON t_user.id = t_order.user_id LIMIT 2, 2
#是对大表做了分页，此时数据可能不完整，比如用户有10个订单，却只能查到2个，或部分。
```

2. 重要提示

`PageHelper.startPage` 方法重要提示

只有紧跟在 `PageHelper.startPage` 方法后的**第一个**Mybatis的**查询（Select）**方法会被分页。

请不要配置多个分页插件

请不要在系统中配置多个分页插件(使用Spring时，`mybatis-config.xml` 和 `Spring<bean>` 配置方式，请选择其中一种，不要同时配置多个分页插件)!

分页插件不支持带有 `for update` 语句的分页

对于带有 `for update` 的sql，会抛出运行时异常，对于这样的sql建议手动分页，毕竟这样的sql需要重视。